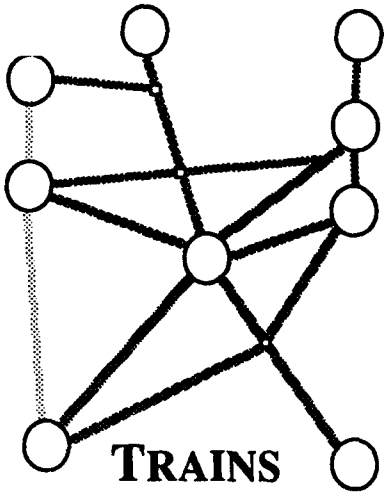
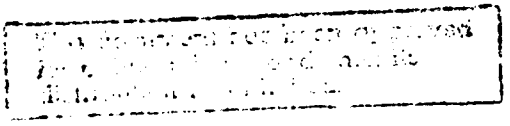


Domain Plan Reasoning in TRAINS-90

George Ferguson

TRAINS Technical Note 91-2
June 1991

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE



Domain Plan Reasoning in TRAINS-90

George Ferguson

The University of Rochester
Computer Science Department
Rochester, New York 14627

TRAINS Technical Note 91-2

June 1991

Abstract

This report describes the domain plan reasoning aspects of the TRAINS-90 project, an integrated, discourse-oriented system for natural language and planning research. The domain planner provides an interface which is used by higher level (*i.e.*, discourse level) routines to address issues that require reasoning about the world. In particular, it must support both planning and plan recognition, since these tasks are interleaved in a discourse-driven system such as this. A uniform representation for both tasks is used, based on a declarative, hierarchical description of actions and plans. Heuristic rules for both planning and plan recognition are also represented declaratively. Descriptions of the system as a whole and of the domain reasoner in particular are presented, as well as documentation of the domain reasoner source code.

This material is based upon work supported by ONR/DARPA under Research Contract number N00014-90-J-1811, Air Force - Rome Lab under Research Contract no. F30602-91-C-0010, and the National Science Foundation under Grant number IRI-9003841. The Government has certain rights in this material.

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TRAINS TN 91-2	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Domain Plan Reasoning in TRAINS-90		5. TYPE OF REPORT & PERIOD COVERED Technical Note 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) George Ferguson		8. CONTRACT OR GRANT NUMBER(s) N00014-90-J-1811
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Department 734 Computer Studies Bldg. University of Rochester, Rochester, NY 14627, USA		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE June 1991 13. NUMBER OF PAGES 26 pages
14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)* Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. *(of this report)* unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

planning; plan recognition; knowledge representation; declarative representations; discourse-driven systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report describes the domain plan reasoning aspects of the TRAINS-90 project, an integrated, discourse-oriented system for natural language and planning research. The domain planner provides an interface which is used by high level (discourse level) routines to address issues that require reasoning about the world. In particular, it must support both planning and plan recognition, since these tasks are interleaved in a discourse-driven system such as this. A uniform representation for both tasks is used, based on a declarative,

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

20. ABSTRACT (Continued)

hierarchical description of actions and plans. Heuristic rules for both planning and plan recognition are also represented declaratively. Descriptions of the system as a whole and of the domain reasoner in particular are presented, as well as documentation of the domain reasoner source code.

1 Introduction

This report describes the domain reasoning aspects of the TRAINS-90 project, an integrated, discourse-oriented system for natural language and planning research. The domain of discourse is the transportation world, with factories, warehouses, and trains all constituting separate, autonomous agents. The model is one of a human user being aided in formulating plans by an intelligent assistant (the system), and then interacting through the system with a simulated world. In addition, the system might have goals of its own, such as scheduled deliveries, and must reason about conflict and trade-offs. The aim of the initial phase of the project being reported here and by others (see [Allen and Schubert, 1991] for a complete list of related publications) was to develop a working system which future, more in-depth, research problems could use as a foundation. In this respect the system can be considered a success, since a human can in fact sit down in front of it and carry on a dialog. However, in order to develop the system completely within a short time frame,[1] many simplifications had to be made. That is, the human would rapidly run out of patience with the system. Nonetheless, the system was designed to be extendible, with modules whose subtleties were glossed easily replaceable.

Development took place on Sun workstations and Symbolics Lisp machines. The system is implemented in Common Lisp and, except for the world simulator which uses Symbolics specialties, would run in any Common Lisp environment. The domain reasoner itself is written in a PROLOG dialect that is interpreted by a Lisp interpreter developed specially for the project.

The next section describes the TRAINS-90 project in more detail, illustrating the position of the domain reasoner in the overall system organization. The third section describes the goals we set for the domain reasoner and outlines design decisions and idealizations that we made to accomplish these goals. The fourth and fifth sections of the report describe the implementation of the domain reasoner in detail, focusing on the translation of high-level intuitions about planning and plan recognition into declarative facts and rules. The details of our chosen knowledge representation and inference procedures are also described in these sections. Finally, we present a summary of some of the goals that were not met in the prototype system and offer some additional directions for future research, then summarize our results in the last section. Appendices are included that present sample runs and cross-reference the source code.

2 System Overview

As mentioned in the introduction, the TRAINS-90 project was designed to support research in natural language processing, discourse reasoning, and planning. The system organization, shown in Figure 1, reflects these various aspects.

Natural language input entered on a terminal[2] is fed through a syntactic parser which

[1] The work described here was carried out during the Summer and early Fall of 1990.

[2] Additional research within the TRAINS-90 project is investigating using spoken natural language [Nakajima and Allen, 1991] but the current system accepts only typed input.

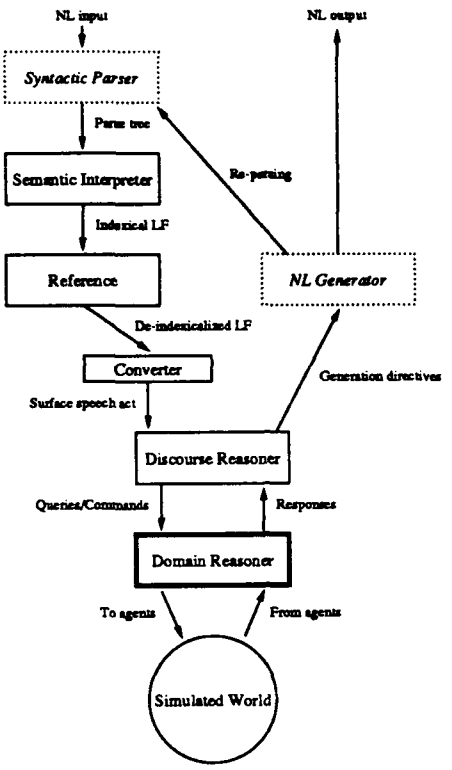


Figure 1: System Organization

generates one or more parse trees corresponding to the preferred parse of the (possibly several) sentences in the input. The parse tree is converted into a surface logical form possibly containing indexical terms; these are eliminated in the third stage which generates a complete, non-indexical logical form for the input. For more details on the current state of the natural language parts of the system, see [Schubert, 1991; Light, 1991].

Due to the fact that the natural language and planning parts of the system were developed in parallel by separate groups, a small pattern-matching syntactic converter transforms the de-indexicalized logical form into a set of surface speech acts. These are processed by the discourse reasoner (see [Traum, 1991]) which is responsable for maintaining the coversation and accomplishing any goals that it is given. It maintains a set of belief spaces corresponding to different types of user and system knowledge about the conversation. The discourse reasoner can interact with the domain reasoner to determine the relevance of a particular speech act, can generate speech acts based on responses from the domain reasoner (or otherwise) by sending commands to a natural language generator, or can instruct the domain planner to attempt various operations such as refining or dispatching a plan. The rest of this report deals entirely with the domain reasoner.

3 Domain Plan Reasoner Goals and Responsabilities

The domain plan reasoner provides an interface that is used by higher level (*i.e.*, discourse level) routines to address issues that require reasoning about the objects and agents in the transportation world. One of the main goals of the project was to develop a representation which could support both planning and plan recognition, since both of these are essential within a discourse-driven system. Plan recognition is used to understand the content of the user's utterances, primarily in order to achieve the correct plan with respect to the their intentions. It is also used however for disambiguation and reference, when linguistic information is not sufficient. The planning aspect of the system is more traditionally understood, namely it constitutes the "assistant" part of the system's functions, completing and checking plans before executing them. Both aspects can be used by the discourse module to guide the conversation. Plan recognition can point out ambiguities in the input which require that clarification sub-dialogs be initiated, or can provide additional information about the plan as recognized that the discourse module can use to provide "helpful" suggestions. Planning can indicate points in the plan where the system cannot proceed, again necessitating additional interaction with the user, or can indicate that a plan is believed to be executable.

This interleaving of planning and plan recognition raises some interesting issues not normally addressed by systems which only have to do one or the other task, and only do it once. As indicated above, ambiguities and choices can be postponed in the hopes that further interaction with the human will provide sufficient details to make the choice. As well, since the domain planner is in some sense a slave to the discourse reasoner, it must be prepared to be called with an almost arbitrary sequence of requests. The implementation cannot make any assumptions that it cannot later retract. The discourse-driven nature of the project also places constraints on the amount of time taken by the domain reasoner in any of its roles. The discourse system must keep the conversation going while the plan reasoner computes, or might need to interrupt or restart it if new information arrives. While this last has not been addressed in the prototype system, it remains an important area for future system development.

Another main goal of the domain reasoner design was to explicitly represent planning mechanisms. Thus the declarative knowledge base (described in the next section) describes exactly how different types of statements from the human should be connected to the current plan. Further, they describe how to complete the plan and test various aspects of it. These are, of course, meta-level rules, in addition to the object-level rules about properties of agents and objects in the transportation world and their relationships.

Main research issues to be addressed by the domain reasoner include using a temporally explicit representation for reasoning about action and change (see, *e.g.*, [Allen, 1984]), and using a representation that allows reasoning in the presence of uncertainty. Both requirements are necessary for any complicated reasoning about interacting plans and independent agents. The latter requirement is due to (a) the fact that the system does not have complete knowledge of the world, but only what it obtains (at some cost) from the agents, and (b) given this realistic restriction, it must reason about persistence of information that it has gathered. Certainly probability models provide a suitable approach to the former problem,

and nonmonotonic formalisms (including probabilistic ones) to the latter.

Finally, the domain reasoner must be agent-oriented. That it, it must support reasoning about multiple agents with different capabilities, and multiple levels of abstractions corresponding to different ways of instructing these agents. New agents and changing capabilities of old agents must be able to be accomodated in a straightforward manner. It will be seen that the chosen representation allows exactly this type abstraction.

4 Domain Knowledge Representation

In this section we describe our approach to representing the transportation world (the object level) and the domain reasoner's capabilities (the meta level). Both types of knowledge are expressed declaratively, using a PROLOG interpreter developed especially for the project. In this section we describe how this language is used to represent the domain reasoner's knowledge. In the next section we describe how this knowledge is used. Further details of the PROLOG interpreter language and implementation are available in an appendix.

4.1 Domain knowledge representation

The following syntax is used for our domain knowledge representation:

- Variables are names starting with a question mark, *i.e.*, `?x`, `?y`, *etc.* The anonymous variable `?_` can be used as well.
- Terms are represented by lists, *i.e.*, (`functor args...`). The standard PROLOG use of square brackets for lists is unnecessary; Lisp lists are used.
- Atomic formulas are also lists, *i.e.*, (`predicate terms...`). The first element of the list is the principal functor of the formula.
- Rules (clauses) are a list of atomic formulas; the first element is the head (consequent) of the rule and the remaining elements are the body (antecedents). The input parser accepts various syntactic variants of these such as the standard PROLOG notation `head :- body` for rules.

Note that case is not significant in Common Lisp, and hence not in our PROLOG either. Certain built-in and library predicates are used; they will be described as necessary.

Using this language, we had to represent two types of knowledge about the world: background information and action descriptions. First, background information and information about the current state of the world were represented in a straightforward, first-order way. For the sample system, the background knowledge is very simple, amounting to a description of the initial location of objects in the world:

```
(at (ojfactory f1 ?_) (city cityB ?_)).
(at (car c1 ?_) (city cityI ?_)).
(at (oranges o1 ?_) (city cityI ?_)).
(at (warehouse w1 ?_) (city cityI ?_)).
```

As can be seen from this code, objects in the world are represented as terms (triples) of the form (`type name parm`). The type field allows us to implement a flat type system without changing the unifier, since terms will only unify with like-typed terms. The name is simply a constant term referring to the object. The parameter field allows us to keep bindings consistent across different clauses when the name of the object is not known. Recall that in PROLOG, variables in different clauses are distinct. However, in planning we may have many clauses (facts) that refer to the same as-yet-unspecified object. Instantiating any one instance ought to instantiate them all. Viewing these as parameters of the plan, we can assign a new constant to the parameters of a new action when the clauses are asserted in order to keep the bindings consistent.[3] Two other approaches are possible. First, we might have used a global symbol table, but this would again require modifications to the unifier. Alternatively, all facts about the plan could be stored as terms in a single clause, but this seemed less persipicuous and less efficient (although whether it actually is or not is not obvious). Other object-level predicates include `coupled`, `in`, and the like, concepts ubiquitous to the transportation domain.

The other type of domain knowledge that we had to represent was the actions available to the planner via the agents in the simulated world. We wanted a representation that could be used for both planning and plan recognition, and that required the minimum commitment to details of other aspects of the reasoner. We chose to represent actions by *action schemas* that would list the major effects, preconditions, and constraints on the actions, as well as describing a hierarchy of actions, including abstract (non-executable) ones, for hierarchical planning. The following is a typical example of a high-level action-schema:

```
(action_schema
        (move_car ?Act ((agent ?E ?P1) (car ?C ?P2)
                        (city ?S ?P3) (city ?D ?P4)))
        (achieves ((at (car ?C ?P2) (city ?D ?P4))))
        (enablers ((at (engine ?E ?P1) (city ?S ?P3))
        (coupled (engine ?E ?P1) (car ?C ?P2))))
        (actions ((move_eng ?MAct ((engine ?E ?P1) (city ?D ?P4)))))
        (constraints ((at (car ?C ?P2) (city ?S ?P3))
                      (not (at (car ?C ?P2) (city ?D ?P4))))))).
```

The functor `action_schema` is the predicate under which this fact is stored in the database. Its argument is a complex term consisting of

(a) the type of action, in this case `move_car`;

(b) an identifier (`?Act`) that will uniquely identify an instance of the schema when bound to a new constant;

(c) a list of the parameters of the action, in the format described above, in this case the agent moving the car, the car being moved, and the source and destination cities;

(d) terms for the various categories of information about the action.

[3]This problem is also addressed in similar way in [Pednault, 1986].

The **achieves** slot[4] describes the main effects of the action, at least those relevant to planning. The **enablers** slot lists preconditions that the domain reasoner should work to achieve, as in backward-chaining. The **actions** slot describes subactions that must be performed to perform this abstract action. In this case, moving a cae requires moving the relevant engine (which is the agent of the **move-car** action). This enforces a certain intensional aspect on plan recognition; simply recognizing pre- and post-conditions of an action is not always sufficient to assert that the action occurred. It also provides the framework for decomposing actions to permit hierarchical planning. Finally, the **constraints** slot represents preconditions that we would typically not work to achieve. These might typically be quantity conditions,[5] such as a relationship between the inputs and outputs of an action, although they are not in this simple example. In effect, this slot and the **enablers** slot allow the encoding of heuristic information for controlling backward-chaining deductive planning.

So in the example, we have described that moving a car ?C from ?S to ?D can be achieved by having an engine ?E at ?S, coupling it to ?C, and moving the engine to ?D. Other actions in the plan (probably higher-level ones) should ensure that the car is already at ?S (to avoid an infinite planning regress) and that the car is not already at ?D (to avoid undoing previous work).

The astute reader will have noticed that this representation is not temporally explicit. In the interests of getting the prototype system running within the time constraints, we were forced to not consider temporal reasoning. Thus actions are sequenced by their preconditions, as will be described later when the dispatching functions are described. This decision also affected some other aspects of the representation; the places where hacks were required for things that would normally be handled by a temporal reasoner will be pointed out as they arise.

Aside from this shortcoming, this declarative representation of states of the world and of actions affords a modular design with a clean first-order semantics. New actions can be added and existing actions modified fairly independently, as our experience showed us during development. The "passive" description of actions also avoids making any committment to how that knowledge might be used for planning or plan recognition, which is the subject of the next section.

5 Domain Plan Reasoning

Since one of the principal goals for the domain reasoner was to represent planning and plan recognition declaratively, we use the same language as before to describe these meta-level operations. The PROLOG predicates **assert** and **retract**, along with **gensym**, are used to modify the database to represent the state of knowledge of the current plan. Three types of knowledge are represented, corresponding to the three main tasks required of the domain reasoner: incorporating an observation into a plan (plan recognition), elaborating a plan (planning), and dispatching a plan, which constitutes interacting with the agents to perform

[4] We borrow the frame representation [Hayes, 1979] terminology here, since the intuitions are very similar.

[5] These are called "resource conditions" by Wilkins [Wilkins, 1988].

the actions specified by the plan. Each of these will be described in turn. Description of the actual linkage between these PROLOG routines and the rest of the system is described later.

5.1 Incorporation

For plan recognition, the domain reasoner must attempt to *incorporate* a given item into a plan. We assume that the discourse module has determined which belief space and plan is relevant, and so it asks the domain reasoner to incorporate the item into the plan, using as its database the appropriate belief space. The method used by the domain reasoner depends on the sort of item it is asked to incorporate. The reasoner returns **(yes)** if the information was already in the plan, a list of items that were added to the plan to incorporate the item, or an indication of why the incorporation failed. The general intuition behind incorporation is that we are trying to make a causal connection between the observation being incorporated and the known plan. There are different ways of making this connection, and these are reflected in the different strategies described below. The system hopes to find a unique shortest connection to the plan. If one cannot be found then an ambiguity is signalled, again as described below.

Plans are described by a set of meta-predicates that describe the current state of knowledge of the plan.[6] The complete list of meta-predicates is given in Table 1.

(plan_goal ?G ?P)	**?G** is a goal of plan **?P**
(plan_action_in ?A ?P)	**?A** *is an action in plan* **?P**
(plan_constraint ?C ?P)	**?C** is a constraint on plan **?P**
(plan_achieves ?A ?E ?P)	**?E** is an effect of action **?A** in plan **?P**
(plan_enabler ?A ?E ?P)	**?E** enables action **?A** in plan **?P**
(plan_subaction ?A ?Aa ?P)	**?Aa** is a subaction of action **?A** in plan **?P**
(plan_act_constraint ?A ?C ?P)	**?C** is a constraint on action **?A** in plan **?P**

Table 1: Plan description meta-predicates

Generally a list of such predicates is returned by the domain reasoner for possible addition to a database by the discourse reasoner. As we will see, these additions are also handled by the domain reasoner in order that it can keep its knowledge base consistent. Note that when we say a value is "returned", we mean that a variable designated as the return value in the query is instantiated, since these are predicates.

If asked to incorporate a *goal* **?G** into a plan **?P**, the domain reasoner will return the clause **(plan_goal ?G ?P)**. In addition, if the goal is the name of an action (as determined from the action schemas), then it is assumed that the action is in the plan, since this is a common intuition. In this case the clause **(plan_action_in ?G ?P)** will also be returned.

Continuing with possible incorporation requests, the domain reasoner may be asked to incorporate the fact that a new *action* is in the plan into the current plan. If the domain

[6] In the current belief space (or database). Everything that follows is relative to the belief space. The mapping from belief space to database is described below when the domain reasoner interface is discussed.

reasoner does not already believe that the action is in the plan (in which case it returns **(yes)**), it must attempt to connect it to actions in the plan through conditions or subactions. Furthermore, it will only do this if there is an unambiguous connection, otherwise it will indicate that there is a choice point in its return value.

The handling of these disjunctive ambiguities is important. In keeping with the discourse-driven nature of the system, their resolution might be postponed, hoping for the arrival of new information. In other cases they have to be resolved immediately, for example when a plan is about to be executed or when a definite answer is required for other reasoning. The approach in the prototype system is to simply indicate the ambiguity to the discourse reasoner by returning a term **(choice term1 term2 ...)** that represents it. In some cases the discourse reasoner makes the choice and subsequently incorporates it back to the domain reasoner, in others it simply waits. Certainly, more sophisticated handling of disjunction and ambiguity is an important area for future research.

In any event, the domain reasoner determines the possible ways of incorporating the action by checking the following two conditions, after first testing whether the action is already in the plan:

(a) The action is a subaction of an action already in the plan, and the constraints of both actions are satisfied;

(b) The action achieves an effect that is an enabler of an action already in the plan, and both actions' constraints are satisfied.

Together, these tests constitute a "one-step" connection to the plan. In general, we might want to reason about arbitrarily long such connections by iterating the one-step ones. For the prototype system however, we simply enumerate the cases that make up two-step connections, since that was sufficient for the demonstration dialog. In theory however, arbitrary connections do not pose a problem, although they do increase the likelihood of ambiguity. The **setof** builtin predicate is used to gather all instances satisfying the tests, and appropriate lists of plan description clauses are returned depending on the instances found. If no such instances are found, then a term of the form **(error no_action)** is returned to indicate that the action simply could not be connected to the plan.

If neither a goal nor an action is being incorporated, then the domain reasoner falls back on its generic strategy to incorporate a *fact*. The reasoner first attempts to incorporate the fact as an enabler of an action that can be incorporated into the plan. Failing that, it attempts to incorporate the fact as a constraint on an action that can be incorporated into the plan. Note that the action used for the connection needn't be in the plan already; the system will return whatever was required to incorporate it in addition to plan description predicates for incorporating the fact itself. One interesting feature arises here with the predicate **one_good_action/2**. This predicate prevents the existence of multiple ways of connecting a fact to the plan from being interpreted as an ambiguity if there is only one shortest such connection.

Table 2 summarizes the possible forms of incorporation requests and their possible results. We believe that these possibilities cover a broad range of phenomena that might be encountered when providing plan recognition services within the transportation domain

Request	Return	Condition
(goal ?G ?P)	(yes)	(plan_goal ?G ?P)
	(plan_goal ?G ?P) (plan_action_in ?A ?P)	?G matches action-schema ?A
	(plan_goal ?G ?P) (plan_constraint ?G ?P)	otherwise
(action_in ?A ?P)	(yes)	(plan_action_in ?A ?P)
	(plan_action_in ?A ?P) (plan_subaction ?Aa ?A ?P)	(plan_action_in ?Aa ?P) and one-step subaction
	(plan_action_in ?A ?P) (plan_achieves ?A ?E ?P) (plan_enabler ?Aa ?E ?P)	(plan_action_in ?Aa ?P) and one-step condition
	(plan_action_in ?A ?P) (plan_subaction ?Aa ?A ?P) (plan_action_in ?Aa ?P) (plan_subaction ?Aaa ?Aa ?P)	(plan_action_in ?Aaa ?P) and two subaction links
	(plan_action_in ?A ?P) (plan_subaction ?Aa ?A ?P) (plan_action_in ?Aa ?P) (plan_achieves ?Aa ?E ?P) (plan_enabler ?Aaa ?E ?P)	(plan_action_in ?Aaa ?P) and one subaction and one condition link
	(plan_action_in ?A ?P) (plan_achieves ?A ?E ?P) (plan_enabler ?Aa ?E ?P) (plan_action_in ?Aa ?P) (plan_subaction ?Aaa ?A ?P)	(plan_action_in ?Aaa ?P) and one subaction and one condition link
	(plan_action_in ?A ?P) (plan_achieves ?A ?E ?P) (plan_enabler ?Aa ?E ?P) (plan_action_in ?Aa ?P) (plan_achieves ?Aa ?Ee ?P) (plan_enabler ?Aaa ?Ee ?P)	(plan_action_in ?Aaa ?P) and two condition links
	(error no_action)	otherwise
(fact ?F ?P)	(yes)	(plan_enabler ?F ?P)
	(yes)	(plan_constraint ?F ?P)
	(plan_enabler ?A ?F ?P) and results of incorporating ?A	?A can be incorporated
	(plan_act_constraint ?A ?F ?P) and results of incorporating ?A	?A can be incorporated
	(plan_constraint ?A ?F ?P)	otherwise

Table 2: Incorporation request and return summary

(and presumably elsewhere also). The declarative rules are clearly and concisely expressed, and would be even more so but for our *ad hoc* treatment of iterated connections.

5.2 Elaboration

The incorporation process described above provides plan recognition facilities to the higher level modules in the TRAINS-90 system. Whenever the discourse module determines that it should take the initiative (usually whenever it gets the "turn," again see [Traum, 1991] for details), it calls the domain reasoner to *elaborate* a plan in a belief space. This corresponds to more standard notions of planning, and is broken into two aspects: elaborating subactions and elaborating preconditions. As with incorporation, a list of plan description predicates is returned and, as opposed to incorporation, they are also asserted to the current belief space. The reason for the difference is that while something that the system plans is definitely known by it, in some sense, the conclusions reached by plan recognition are always in some sense tentative, thus requiring finer discourse-level control.

As we noted when the action-schema representation was described, `subaction` slots provide a means of defining actions hierarchically. The first task of the elaboration phase is to ensure that any subactions that are not already part of the plan are added to it. These new actions are also subject to the elaboration process. As long as the action hierarchy is acyclic (as seems reasonable), no infinite regress is possible.

Besides the action decomposition, the elaboration phase must ensure that all preconditions (`enablers`) are fulfilled. It does this by gathering up all those enablers of actions in the plan that are not already explicitly listed as part of the plan by a `plan_enabler` clause. It removes those which are already true for any reason (thus performing opportunistic planning) and tries to achieve the rest by adding new actions to the plan. These actions are subject to further elaboration, and care must be taken to avoid an infinite regress. The distinction between `enablers` and `constraints` provides a mechanism for controlling the backward chaining. In both cases the method of achieving the enabler must be unambiguous or a choice error will be returned in addition to whatever could be elaborated unambiguously. Interestingly, the test for an enabler already being true can introduce ambiguity, for example when an unbound parameter can be instantiated in several ways.

As might be expected, this simple approach to planning yields far too many ambiguities far too quickly to be of much use as a real planner, although it performs adequately for demonstration purposes. The use of a perspicuous declarative formalism for expressing the planning rules, however, makes it easy to alter and improve the elaboration procedures as part of future work on the system. By way of example, we note that the elaboration module was added to the system after the initial deadline for development, with no problems.

5.3 Dispatching

The final major phase of the domain reasoner's job is to *dispatch* a plan for execution. It does this, as always, under the guidance of the discourse system when that module has determined that the plan is executable (usually after a `nil` elaboration return) and that the user wants it executed. The main reason that this is a separate phase is that the

agents and their simulated world were developed separately (again, see [Martin and Miller. 1991] for details) thereby necessitating certain syntactic transformations. As part of future research however, the domain reasoner's tasks may be interleaved with execution of plans and plan fragments by the agents. It is expected that this dispatching module will contain the routines required to react to changes in the world, and modify plans or inform the discourse module of more serious problems.

For current purposes, the predicate **make_executable_plan/2** simply gathers the primitive actions of the plan (those with no subactions, which will actually be executed) and converts them to *condition-action* pairs for the reactive agents. The conditions for an action are

(a) Any enablers listed as part of the plan;

(b) Any constraint listed in the schemas;

(c) Any preconditions of actions of which this action is a subaction.

The domain reasoner removes any of these that are already true (although it needn't technically). Note that only those enablers that were explicitly recognized as important and therefore described by **plan_enabler** clauses are considered, whereas all constraints are listed. This is essentially an artifact of our poor handling of constraints, although it does not seem to be an incorrect approach.

It is in this phase that our lack of explicit temporal notation is most obvious. Several preconditions required for the sample dialog must be explicitly mentioned in order that they not be removed. That is, the domain reasoner thinks they are already true, although they will really only be true at some point in the future. A temporally explicit representation would not have this problem and the hacked-in cases could be removed.

Finally, the minor syntactic transformations required for the agents are performed and a term representing the plan in executor format is returned.

5.4 Domain Reasoner Interface

Since the rest of the TRAINS-90 system is written in Common Lisp, interface routines are provided that connect to the PROLOG meta-predicates described above. In general these rely on using the **lisp/2** builtin predicate to set a special variable from within PROLOG during a call to **prove**. This variable can then be accessed after the proof has finished and its value used by Lisp routines. The Lisp functions **incorporate**, **elaborate**, and **execute** correspond to the three phases described above. All three accept a plan or item parameter and a belief space in which to operate. They convert the parameter into a goal statement and call **prove_in_prolog** which sets the special variable ***database*** to the given belief space and then calls **prove**.

It is clear from the above that belief spaces are PROLOG databases; the planning meta-predicates are oblivious to them. In general, the discourse system might maintain more sophisticated information in these spaces, but for the prototype system they contain only plan description and object-level predicates. Functions are provided to create, move, and

copy belief spaces, and these also provide the inheritance between the `*shared*` space and all the others. The function `print-plan` pretty-prints plans by proving the PROLOG predicate `printPlan/1` in the given belief space.

6 Future Work

Future work on the TRAINS-90 project domain plan reasoner can be divided into two phases. Initially, those aspects of the systems that were implemented cheaply due to time constraints must be improved. First, we need a real type system within the unifier to provide true hierarchical typing and clarify the language. Secondly, constraints should be posted on variables and propagated during inference, rather than being tested at more or less arbitrary points. Finally, the as the previous discussion clearly illustrated, we must modify the representation to be temporally explicit. None of these developments are seen as major hurdles, and indeed most of this functionality can be found in the RHET system [Allen and Miller, 1989].

Longer-term research in domain plan reasoning will center on the introduction of mechanisms for dealing with uncertainty. This will probably include the introduction of non-monotonic methods for plan reasoning and of probablistic methods for sensor and agent modelling. In particular, the treatment of ambiguity in incorporation and elaboration is insufficient for a real system. We will want to reason more effectively about alternatives before signalling a choice to the discourse reasoner. As well, techniques for plan monitoring, replanning, and the interleaving of planning and execution need to be developed and implemented to replace the simple dispatching routine used currently.

7 Conclusions

We have described the domain reasoning aspects of the TRAINS-90 project in detail. We have described the way this module interacts with the rest of the TRAINS-90 system to the level of Lisp functions. We have also described the intuition behind the various services required of the domain reasoner, and described in some detail how these services are implemented. It is hoped that this report will provide not only a summary of the work done to date on the prototype system, but also a guide to further development. Appendices provide (a) a description of the domain reasoning aspects of the sample dialog, (b) a description of the PROLOG subsystem, and (c) a cross-reference guide to the source code.

References

[Allen and Schubert, 1991] James F. Allen and Lenhart K. Schubert, "The TRAINS Project," TRAINS Technical Note 91-1, Department of Computer Science, University of Rochester, Rochester, NY, 1991.

[Allen, 1984] J.F. Allen, "Towards a general theory of action and time," *Artificial Intelligence*, 23:123–154, 1984, Also in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate (eds.), Morgan Kaufmann, 1990, pp. 464–479.

[Allen and Miller, 1989] J.F. Allen and B.W. Miller, "The Rhetorical knowledge representation system: A user's manual (for Rhet version 15.25)," Technical Report 238 (revised), Department of Computer Science, University of Rochester, Rochester, NY, March 1989.

[Hayes, 1979] P.J. Hayes, "The logic of frames," In D. Metzing, editor, *Frame Conceptions and Text Understanding*, pages 46–61. Walter de Gruyter and Co., Berlin, 1979, Also in *Readings in Knowledge Representation*, R.J. Brachman and H.J. Levesque (eds.), Morgan Kaufmann, 1985, pp. 287–296.

[Light, 1991] Marc Light, "Semantic interpretation in TRAINS-90," TRAINS Technical Note 91-3, Department of Computer Science, University of Rochester, Rochester, NY, 1991.

[Martin and Miller, 1991] Nathaniel Martin and Bradford Miller, "The TRAINS-90 simulator," TRAINS Technical Note 91-4, Department of Computer Science, University of Rochester, Rochester, NY, 1991.

[Nakajima and Allen, 1991] Shinya Nakajima and James F. Allen, "A study of pragmatic roles of prosody in the TRAINSdialogs," Trains technical note, Department of Computer Science, University of Rochester, Rochester, NY, 1991, To appear.

[Nilsson, 1984] M. Nilsson, "The world's shortest Prolog interpreter?," In J.A. Campbell, editor, *Implementations of Prolog*, pages 87–92. Ellis Horwood, Chichester, England, 1984.

[Pednault, 1986] E.P.D. Pednault, "Formulating multi-agent, dynamic-world problems in the classical planning framework," In M.P. Georgeff and A.L. Lansky, editors, *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, Los Altos, CA, 30 June–2 July 1986. Morgan Kaufmann, Also in *Readings in Planning*, J. Allen, J. Hendler, and A. Tate (eds.), Morgan Kaufmann, 1990, pp. 675–710.

[Schubert, 1991] Lenhart K. Schubert, "Language processing in the TRAINSproject," Trains technical note, Department of Computer Science, University of Rochester, Rochester, NY, 1991, To appear.

[Traum, 1991] David Traum, "The discourse reasoner in TRAINS-90," TRAINS Technical Note 91-6, Department of Computer Science, University of Rochester, Rochester, NY, 1991.

[Wilkins, 1988] D.E. Wilkins, *Practical Planning: Extending the Classical AI Planning Paradigm*, Morgan Kaufmann, San Mateo, CA, 1988.

A System Traces

A.1 Sample Dialog

The following is a trace of the domain reasoner's execution on the sample dialog for the prototype system, which was as follows:

> Manager: We have to make OJ.
> Manager: There are oranges at I and an OJ factory at B.
> Manager: Engine E3 is scheduled to arrive at I at 3 pm.
> Manager: Shall we ship the oranges?
> System: Ok.
> System: Shall I start loading the oranges into the empty car at I?
> Manager: Yes, and we'll have E3 pick it up.
> Manager: Ok?
> System: Ok.

In what follows, user input and system natural language output are shown in *italics*, and calls from the discourse reasoner to the domain reasoner and values returned to the discourse reasoner are in `constant-width`. Calls to the domain reasoner use mixed-case while returns from it are entirely uppercase.

Manager: We have to make OJ.

```
(incorporate '(goal (make_oj (:*var* Act) (:*var* Parms)) plan1) *hprop*)

((PLAN_GOAL
     (MAKE_OJ #:ACT1062
              ((OJFACTORY (:*VAR* F) #:PARM1063)
               (ORANGES (:*VAR* O) #:PARM1064)
               (OJ (:*VAR* OJ) #:PARM1065)
               (CITY (:*VAR* L) #:PARM1066)))
     PLAN1)
 (PLAN_ACTION_IN
     (MAKE_OJ #:ACT1062
              ((OJFACTORY (:*VAR* F) #:PARM1063)
               (ORANGES (:*VAR* O) #:PARM1064)
               (OJ (:*VAR* OJ) #:PARM1065)
               (CITY (:*VAR* L) #:PARM1066)))
     PLAN1))
```

Manager: There are oranges at I and an OJ factory at B.

```
(incorporate '(fact (at (oranges o1 (:*var* _1))
                        (city cityI (:*var* _2))) plan1) *hprop*)
```

```
((PLAN_ACT_CONSTRAINT
     (MOVE_ORANGES #:ACT1067
         ((AGENT (:*VAR* A) #:PARM1068) (ORANGES O1 #:PARM1064)
          (CAR (:*VAR* C) #:PARM1069) (CITY CITYI #:PARM1070)
          (OJFACTORY (:*VAR* F) #:PARM1063)))
     (AT (ORANGES O1 #:PARM1064) (CITY CITYI #:PARM1070)) PLAN1)
 (PLAN_ACTION_IN
     (MOVE_ORANGES #:ACT1067
         ((AGENT (:*VAR* A) #:PARM1068) (ORANGES O1 #:PARM1064)
          (CAR (:*VAR* C) #:PARM1069) (CITY CITYI #:PARM1070)
          (OJFACTORY (:*VAR* F) #:PARM1063)))
     PLAN1)
 (PLAN_ACHIEVES
     (MOVE_ORANGES #:ACT1067
         ((AGENT (:*VAR* A) #:PARM1068) (ORANGES O1 #:PARM1064)
          (CAR (:*VAR* C) #:PARM1069) (CITY CITYI #:PARM1070)
          (OJFACTORY (:*VAR* F) #:PARM1063)))
     (AT (ORANGES O1 #:PARM1064) (OJFACTORY (:*VAR* F) #:PARM1063))
     PLAN1)
 (PLAN_ENABLER
     (MAKE_OJ #:ACT1062
              ((OJFACTORY (:*VAR* F) #:PARM1063)
               (ORANGES O1 #:PARM1064) (OJ (:*VAR* OJ) #:PARM1065)
               (CITY (:*VAR* L) #:PARM1066)))
     (AT (ORANGES O1 #:PARM1064) (OJFACTORY (:*VAR* F) #:PARM1063))
     PLAN1))

(incorporate '(fact (at (ojfactory f1 (:*var* _1))
                        (city cityB (:*var* _2))) plan1) *hprop*)

((PLAN_ACT_CONSTRAINT
     (MAKE_OJ #:ACT1062
              ((OJFACTORY F1 #:PARM1063) (ORANGES O1 #:PARM1064)
               (OJ (:*VAR* OJ) #:PARM1065) (CITY CITYB #:PARM1066)))
     (AT (OJFACTORY F1 #:PARM1063) (CITY CITYB #:PARM1066)) PLAN1)
 YES)
```

Manager: Engine E3 is scheduled to arrive at I at 3 pm.

```
(incorporate '(fact (at (engine eng3 (:*var* _1))
                        (city cityI (:*var* _2))) plan1) *hprop*)

((PLAN_ENABLER
     (MOVE_CAR #:ACT1071
         ((AGENT ENG3 #:PARM1068) (CAR (:*VAR* C) #:PARM1069)
          (CITY CITYI #:PARM1070) (CITY (FACTORY-CITY F1) #:PARM1063)))
     (AT (ENGINE ENG3 #:PARM1068) (CITY CITYI #:PARM1070)) PLAN1)
 (PLAN_ACTION_IN
     (MOVE_CAR #:ACT1071
         ((AGENT ENG3 #:PARM1068) (CAR (:*VAR* C) #:PARM1069)
```

```
         (CITY CITYI #:PARM1070) (CITY (FACTORY-CITY F1) #:PARM1063)))
     PLAN1)
 (PLAN_SUBACTION
     (MOVE_ORANGES #:ACT1067
         ((AGENT ENG3 #:PARM1068) (ORANGES O1 #:PARM1064)
          (CAR (:*VAR* C) #:PARM1069) (CITY CITYI #:PARM1070)
          (OJFACTORY F1 #:PARM1063)))
     (MOVE_CAR #:ACT1071
         ((AGENT ENG3 #:PARM1068) (CAR (:*VAR* C) #:PARM1069)
          (CITY CITYI #:PARM1070) (CITY (FACTORY-CITY F1) #:PARM1063)))
     PLAN1))
```

Manager: Shall we ship the oranges?

```
(incorporate '(action_in (move_oranges (:*var* Act)
                                ((agent (:*var* _1) (:*var* _2))
                                 (oranges o1 (:*var* _3))
                                 (car (:*var* _4) (:*var* _5))
                                 (city (:*var* _6) (:*var* _7))
                                 (ojfactory (:*var* _8) (:*var* _9)))) plan1)
              *hprop*)

(YES)

(move-plan 'plan1 *hprop* *shared*)

T

(elaborate 'plan1 *splan*)

((PLAN_ACTION_IN
     (MOVE_ENG #:ACT1072
         ((ENGINE ENG3 #:PARM1068) (CITY (FACTORY-CITY F1) #:PARM1063)))
     PLAN1)
 (PLAN_SUBACTION
     (MOVE_CAR #:ACT1071
         ((AGENT ENG3 #:PARM1068) (CAR (:*VAR* C) #:PARM1069)
          (CITY CITYI #:PARM1070) (CITY (FACTORY-CITY F1) #:PARM1063)))
     (MOVE_ENG #:ACT1072
         ((ENGINE ENG3 #:PARM1068) (CITY (FACTORY-CITY F1) #:PARM1063)))
     PLAN1)
 (PLAN_ACTION_IN (RUN #:ACT1073 ((OJFACTORY F1 #:PARM1063))) PLAN1)
 (PLAN_SUBACTION
     (MAKE_OJ #:ACT1062
              ((OJFACTORY F1 #:PARM1063) (ORANGES O1 #:PARM1064)
               (OJ (:*VAR* OJ) #:PARM1065) (CITY CITYB #:PARM1066)))
     (RUN #:ACT1073 ((OJFACTORY F1 #:PARM1063))) PLAN1)
 (PLAN_ACTION_IN
     (UNLOAD_ORANGES #:ACT1074
```

```
        ((OJFACTORY F1 #:PARM1063) (CAR (:*VAR* C) #:PARM1069)
         (CITY CITYB #:PARM1066) (ORANGES O1 #:PARM1064)))
    PLAN1)
(PLAN_ACHIEVES
    (UNLOAD_ORANGES #:ACT1074
        ((OJFACTORY F1 #:PARM1063) (CAR (:*VAR* C) #:PARM1069)
         (CITY CITYB #:PARM1066) (ORANGES O1 #:PARM1064)))
    (HAS_ORANGES (OJFACTORY F1 #:PARM1063)) PLAN1)
(PLAN_ENABLER (RUN #:ACT1073 ((OJFACTORY F1 #:PARM1063)))
    (HAS_ORANGES (OJFACTORY F1 #:PARM1063)) PLAN1)
(PLAN_ACTION_IN
    (LOAD_ORANGES #:ACT1075
        ((CAR (:*VAR* C) #:PARM1069) (ORANGES O1 #:PARM1064)
         (CITY CITYI #:PARM1070)))
    PLAN1)
(PLAN_ACHIEVES
    (LOAD_ORANGES #:ACT1075
        ((CAR (:*VAR* C) #:PARM1069) (ORANGES O1 #:PARM1064)
         (CITY CITYI #:PARM1070)))
    (IN (CAR (:*VAR* C) #:PARM1069) (ORANGES O1 #:PARM1064)) PLAN1)
(PLAN_ENABLER
    (MOVE_ORANGES #:ACT1067
        ((AGENT ENG3 #:PARM1068) (ORANGES O1 #:PARM1064)
         (CAR (:*VAR* C) #:PARM1069) (CITY CITYI #:PARM1070)
         (OJFACTORY F1 #:PARM1063)))
    (IN (CAR (:*VAR* C) #:PARM1069) (ORANGES O1 #:PARM1064)) PLAN1)
(CHOICE ((AT (CAR C2 #:PARM1069) (CITY CITYI #:PARM1070))
         (AT (CAR C1 #:PARM1069) (CITY CITYI #:PARM1070)))))

(move-plan 'plan1 *splan* *sprop*)

T
```

System: Ok. Shall I start loading the oranges into the empty car at I?

```
(incorporate '(fact (at (car c1 (:*var* _1)) (city cityI (:*var* _2)))
                    plan1) *sprop*)

((PLAN_ENABLER
    (MOVE_ORANGES #:ACT1067
        ((AGENT ENG3 #:PARM1068) (ORANGES O1 #:PARM1064)
         (CAR C1 #:PARM1069) (CITY CITYI #:PARM1070)
         (OJFACTORY F1 #:PARM1063)))
    (AT (CAR C1 #:PARM1069) (CITY CITYI #:PARM1070)) PLAN1)
 YES)
```

Yes, and we'll have E3 pick it up.

```
(move-plan 'plan1 *sprop* *shared*)
```

```
T

(incorporate '(action_in (couple (:*var* Act)
                                 ((engine eng3 (:*var* _1)) (car c1 (:*var* _2))
                                  (city (:*var* _3) (:*var* _4)))) plan1)
                 *hprop*)

((PLAN_ACTION_IN
     (COUPLE #:ACT1076
             ((ENGINE ENG3 #:PARM1068) (CAR C1 #:PARM1069)
              (CITY (:*VAR* L) #:PARM1077)))
     PLAN1)
 (PLAN_ACHIEVES
     (COUPLE #:ACT1076
             ((ENGINE ENG3 #:PARM1068) (CAR C1 #:PARM1069)
              (CITY (:*VAR* L) #:PARM1077)))
     (COUPLED (ENGINE ENG3 #:PARM1068) (CAR C1 #:PARM1069)) PLAN1)
 (PLAN_ENABLER
     (MOVE_CAR #:ACT1071
         ((AGENT ENG3 #:PARM1068) (CAR C1 #:PARM1069)
          (CITY CITYI #:PARM1070) (CITY (FACTORY-CITY F1) #:PARM1063)))
     (COUPLED (ENGINE ENG3 #:PARM1068) (CAR C1 #:PARM1069)) PLAN1))
```

Manager: Ok?

```
(move-plan 'plan1 *hprop* *shared*)

T

(elaborate 'plan1 *splan*)

NIL

(execute 'plan1 *splan*)

((((AT ENG3 CITYI)) (COUPLE ENG3 ENG3 C1))
 (((IN C1 O1) (AT C1 CITYB)) (UNLOAD F1 C1))
 (NIL (LOAD W1 C1 O1))
 (((IN C1 O1) (AT C2 CITYI) (COUPLED ENG3 C1) (AT ENG3 CITYI))
  (GOTO ENG3 CITYB))
 (((HAS_ORANGES F1)) (RUN F1)))
```

A.2 Alternate Dialog

In addition to the dialog described above, the user might not accept the system's suggestion to use the car at city I (C1), and suggest instead that the car attached to E3 (C2) be used instead. This leads to a different sequence of elaborates, and of course to a different plan. Since the dialogs are identical up to the point following the system's utterance, we do not repeat the initial part of the trace.

Manager: No, use the empty car at city I instead.

```
(incorporate '(action_in (load_oranges (:*var* Act)
                                ((car c2 (:*var* _1)) (oranges o1 (:*var* _2))
                                 (city (:*var* _3) (:*var* _4)))) plan1)
               *hprop*)

((PLAN_ACTION_IN
     (LOAD_ORANGES #:ACT1076
         ((CAR C2 #:PARM1069) (ORANGES O1 #:PARM1064)
          (CITY (:*VAR* L) #:PARM1077)))
     PLAN1)
 (PLAN_ACHIEVES
     (LOAD_ORANGES #:ACT1076
         ((CAR C2 #:PARM1069) (ORANGES O1 #:PARM1064)
          (CITY (:*VAR* L) #:PARM1077)))
     (IN (CAR C2 #:PARM1069) (ORANGES O1 #:PARM1064)) PLAN1)
 (PLAN_ENABLER
     (MOVE_ORANGES #:ACT1067
         ((AGENT ENG3 #:PARM1068) (ORANGES O1 #:PARM1064)
          (CAR C2 #:PARM1069) (CITY CITYI #:PARM1070)
          (OJFACTORY F1 #:PARM1063)))
     (IN (CAR C2 #:PARM1069) (ORANGES O1 #:PARM1064)) PLAN1))
```

Manager: Ok?

```
(move-plan 'plan1 *hprop* *shared*)

T

(elaborate 'plan1 *splan*)

((PLAN_ACTION_IN
     (MOVE_ENG #:ACT1078
         ((ENGINE ENG3 #:PARM1068) (CITY (FACTORY-CITY F1) #:PARM1063)))
     PLAN1)
 (PLAN_SUBACTION
     (MOVE_CAR #:ACT1071
         ((AGENT ENG3 #:PARM1068) (CAR C2 #:PARM1069)
          (CITY CITYI #:PARM1070) (CITY (FACTORY-CITY F1) #:PARM1063)))
     (MOVE_ENG #:ACT1078
         ((ENGINE ENG3 #:PARM1068) (CITY (FACTORY-CITY F1) #:PARM1063)))
     PLAN1)
 (PLAN_ACTION_IN (RUN #:ACT1079 ((OJFACTORY F1 #:PARM1063))) PLAN1)
 (PLAN_SUBACTION
     (MAKE_OJ #:ACT1062
              ((OJFACTORY F1 #:PARM1063) (ORANGES O1 #:PARM1064)
               (OJ (:*VAR* OJ) #:PARM1065) (CITY CITYB #:PARM1066)))
     (RUN #:ACT1079 ((OJFACTORY F1 #:PARM1063))) PLAN1)
 (PLAN_ACTION_IN
     (UNLOAD_ORANGES #:ACT1080
         ((OJFACTORY F1 #:PARM1063) (CAR C2 #:PARM1069)
```

```
          (CITY CITYB #:PARM1066) (ORANGES O1 #:PARM1064)))
     PLAN1)
 (PLAN_ACHIEVES
     (UNLOAD_ORANGES #:ACT1080
         ((OJFACTORY F1 #:PARM1063) (CAR C2 #:PARM1069)
          (CITY CITYB #:PARM1066) (ORANGES O1 #:PARM1064)))
     (HAS_ORANGES (OJFACTORY F1 #:PARM1063)) PLAN1)
 (PLAN_ENABLER (RUN #:ACT1079 ((OJFACTORY F1 #:PARM1063)))
     (HAS_ORANGES (OJFACTORY F1 #:PARM1063)) PLAN1))

(execute 'plan1 *splan*)

((((IN C2 O1) (AT C2 CITYB)) (UNLOAD F1 C2))
 (((HAS_ORANGES F1)) (RUN F1))
 (((AT C2 CITYI)) (LOAD W1 C2 O1))
 (((IN C2 O1) (AT C2 CITYI) (AT ENG3 CITYI)) (GOTO ENG3 CITYB)))
```

B Prolog Interpreter Description

Our implementation of PROLOG is a simple one, based on that described in [Nilsson, 1984]. The decision to build our own interpreter was simply motivated by the desire to axiomatize the domain and the domain reasoner declaratively and the fact that we did not have such a tool for our Lisp machine environment. Certainly many aspects of the implementation are sub-optimal. However, any other PROLOG subsystem could be substituted with only a change in syntax if necessary.

B.1 Interpreter Description

The following syntax is used:

- Variables are names starting with a question mark, *i.e.*, `?x`, `?y`, *etc.* The anonymous variable `?_` can be used as well.
- Terms are represented by lists, *i.e.*, `(functor args...)`. The standard PROLOG use of square brackets for lists is unnecessary; Lisp lists are used.
- Atomic formulas are also lists, *i.e.*, `(predicate terms...)`. The first element of the list is the principal functor of the formula.
- Rules (clauses) are a list of atomic formulas: the first element is the head (consequent) of the rule and the remaining elements are the body (antecedents). The input parser accepts various syntactic variants of these such as the standard PROLOG notation `head :- body` for rules.

Note that case is not significant in Common Lisp, and hence not in our PROLOG either.

The following evaluable predicates are provided as built-in, since they require access to the unification mechanism:

`(cut)` The standard cut predicate, which always succeeds and removes choice points for the parent goal.

`(bagof ?x ?y ?z)` True if `?z` is the list ("bag") of `?x`'s such that `?y` is provable.

`(lisp ?x ?y)` True if `?y` unifies with the result of evaluating `?x` in Lisp.

`(retract ?x)` True if `?x` matches a clause in the database. As a side-effect the clause is removed from the database. Backtracking into this predicate will remove successive clauses.

The following evaluable predicates are provided as library predicates:

Control Predicates

`(true)` Always succeeds (once).

`(call ?x)` True if `?x` can be proved.

`(not ?x)` True if `?x` cannot be proved.

`(and ?x ?y)` True if both `?x` and `?y` can be proved.

Meta-level Predicates

`(= ?x ?y)` True if `?x` unifies with `?y`.

`(var ?x)` True if `?x` is currently bound to a variable.

`(gensym ?x ?y)` True if `?y` unifies with a new symbol created by calling `(gensym ?x)` in Lisp.

I/O Predicates

(write ?x) Always succeeds. Writes **?x** to the the default output stream as a side-effect.

(nl) Always succeeds. Writes a newline to the the default output stream as a side-effect.

(writeln ?x) Equivalent to **(and (write ?x) (nl))**.

Database Predicates

(assertz ?x) Always succeeds (once). Adds the clause **?x** to the database as a side-effect, at the end of the list of clauses for its head formula.

(asserta ?x) Like **assertz** but adds **?x** at the beginning of the list of clauses for its head formula.

(assert ?x) Equivalent to **(assertz ?x)**.

(consult ?x) Always succeeds. The file given by **?x** (which should be a string) is read and the clauses in it asserted to the database.

(listing) Always succeeds. The database is dumped to the default output stream.

(listing ?x) Always succeeds. Only clauses that are have heads whose principal functor is **?x** are dumped.

Utility predicates

(length ?l ?n) True if the length of the list **?l** is **?n**.

(member ?l ?x) True if **?x** is a member of the list **?l**.

(member ?l1 ?l2 ?l3) True if **?l3** is the result of appending **?l2** to the end of **?l1**.

B.2 Interpreter Implementation

The PROLOG database is implemented as a hash table using the special variable ***database***, which should be **let** to the appropriate database. An empty database can be created with the **new-db** function. An existing one can be cleared or copied using **clear-db** or **copy-db** respectively. The function **init-db** resets the given database to a state where it contains only the PROLOG builtins and library functions.

The main function provided by the interpreter package is the function **prove** which, given a list of goals attempts to prove them all. For each successful proof branch, it calls the function given by ***success-func***, passing the current binding environment as the only parameter. This function should return **nil** to force backtracking, or **t** to not try alternate proofs. Typically ***success-func*** is **let** to a function that prints the top-level bindings and prompts the user for backtracking. During a proof of **setof/3** however, it is **let** to a function that gathers up the instances of **?x** for later return. **prove** returns **t** or **nil** depending on whether the last proof succeeded or failed.

For interactive use, the interpreter package provides the function **prolog**, that implements the "read-prove-print" loop of a standard PROLOG. It returns when **(halt)** is given as the goal to prove. This function and **consult** use the function **read-clause** to implement a simple PROLOG parser. Clauses are terminated with a period. Either ampersands or commas may be used to separate conjuncts. Either **:-** or **<-** may be used to separate the heads and bodies of rules.

This set of functions provides both interactive and non-interactive interfaces to the PROLOG interpreter. The former is used by the functions that implement the linkage between the domain reasoner and the rest of the TRAINS-90 system. The interactive mode is useful for debugging since it allows the database to be examined and queried without reloading or changing the system.

C System Cross-reference

In the following table, V indicates a special variable, M a Lisp macro, F a Lisp function, and P a PROLOG predicate. The third column indicates the source file containing the definition of the construct.

achieveEnabler/4	P	elaborate.lp
achievedEnabler/5	P	elaborate.lp
achieves/2	P	actions.lp
action_in/2	P	actions.lp
action_schema/5	P	actions.lp
(add-to-db rule db)	F	prolog.lsp
(add-to-db-a rule db)	F	prolog.lsp
(add-to-db-z rule db)	F	prolog.lsp
(assert-plan-item item space)	F	planner.lsp
assertItem/1	P	database.lp
assertNewActions/2	P	database.lp
assertNonActions/1	P	database.lp
assertPlanItems/1	P	database.lp
(bagof to-prove nlist env p)	F	prolog.lsp
bagof-env	V	prolog.lsp
bagof-list	V	prolog.lsp
(bagof-topfun env)	F	prolog.lsp
(bind x y e)	M	prolog.lsp
bindParms/1	P	database.lp
(bond x y)	M	prolog.lsp
(but-first-goal x)	M	prolog.lsp
(check-rule rule)	F	prolog.lsp
check2/2	P	incorporate.lp
checkParent/4	P	dispatch.lp
check_constraints/1	P	incorporate.lp
(clear-db db)	F	prolog.lsp
clearPlan/1	P	database.lp
constraint/2	P	actions.lp
(consult filename)	F	prolog.lsp
(copy-db src dst)	F	prolog.lsp
(cut to-prove nlist env n)	F	prolog.lsp
database	V	prolog.lsp
distributeBindings/1	P	database.lp
(elaborate plan space)	F	planner.lsp
elaborate/2	P	elaborate.lp
elaborateEnablers/2	P	elaborate.lp
elaborateSubactions/2	P	elaborate.lp
enabler/2	P	actions.lp
(execute plan space)	F	planner.lsp
filterAct/2	P	dispatch.lp
filterArg/2	P	dispatch.lp
filterCond/2	P	dispatch.lp
filterConds/2	P	dispatch.lp
filterForExecutor/4	P	dispatch.lp
findall/3	P	incorporate.lp

fulfillAll/3	P	elaborate.lp
fulfillEnabler/3	P	elaborate.lp
fulfilledEnabler/5	P	elaborate.lp
gather/3	P	dispatch.lp
gatherAll/3	P	dispatch.lp
hack-parm/2	P	elaborate.lp
hack-parms/2	P	elaborate.lp
hprop	V	planner.lsp
incorp/2	P	incorporate.lp
incorp_action/4	P	incorporate.lp
incorp_action11_p/4	P	incorporate.lp
incorp_action12_p/4	P	incorporate.lp
incorp_action1_p/4	P	incorporate.lp
incorp_action2/4	P	incorporate.lp
incorp_action2_p/6	P	incorporate.lp
incorp_constraint/4	P	incorporate.lp
incorp_constraint_p/4	P	incorporate.lp
incorp_enable_p/4	P	incorporate.lp
incorp_enabler/4	P	incorporate.lp
(incorporate item space)	F	planner.lsp
(init-builtins db)	F	prolog.lsp
(init-db)	F	prolog.lsp
(init-library db)	F	prolog.lsp
(init-planner)	F	planner.lsp
(init-prolog)	F	prolog.lsp
(inst p env)	F	prolog.lsp
(lisp to-prove nlist env p)	F	prolog.lsp
listifySubactions/3	P	elaborate.lp
(lookup p env)	F	prolog.lsp
lp-directory	V	planner.lsp
(lvl x)	M	prolog.lsp
makePrintableAction/2	P	database.lp
makePrintableFormula/2	P	database.lp
makePrintableParm/2	P	database.lp
makePrintableParms/2	P	database.lp
make_executable_plan/2	P	dispatch.lp
(molec x y)	M	prolog.lsp
(move-plan plan src dst)	F	planner.lsp
my_assert/1	P	database.lp
my_retract/1	P	database.lp
(new-db db)	F	prolog.lsp
one_good_action/2	P	incorporate.lp
plan_item/1	P	database.lp
planner-verbose	V	planner.lsp
(print-db db &test)	F	prolog.lsp
(print-plan plan space)	F	planner.lsp
printAction/1	P	database.lp
printFormula/1	P	database.lp
printPlan/1	P	database.lp
(prolog)	F	prolog.lsp
prolog-return	V	planner.lsp

(prove goals)	F	prolog.lsp
(prove-in-prolog clause space)	F	planner.lsp
proveAll/1	P	incorporate.lp
(read-clause &stream ...)	F	prolog.lsp
removeTrueConds/2	P	dispatch.lp
requiredSubaction/3	P	elaborate.lp
(retract to-prove nlist env p)	F	prolog.lsp
schema_achieves/2	P	actions.lp
schema_action_in/2	P	actions.lp
schema_constraint/2	P	actions.lp
schema_enabler/2	P	actions.lp
(seek to-prove nlist env n)	F	prolog.lsp
shared	V	planner.lsp
(show env)	F	prolog.lsp
(show-all term env)	F	prolog.lsp
(space-name space)	F	planner.lsp
splan	V	planner.lsp
sprop	V	planner.lsp
success-func	V	prolog.lsp
unfulfilledEnabler/3	P	elaborate.lp
(unify x y env)	F	prolog.lsp
updateFact/1	P	incorporate.lp
(variablep item)	F	prolog.lsp
(variablize formula)	F	prolog.lsp
(xpr x)	M	prolog.lsp